



Relatório Técnico

**Núcleo de
Computação Eletrônica**

Data Encryption Standard Simulation and a Bit-Slice Architecture Design

**R. G. Sixel
R. S. Monteiro
M. L. Anido**

NCE - 17/2000

Universidade Federal do Rio de Janeiro

Data Encryption Standard Simulation and a Bit-Slice Architecture Design

R. G. Sixel, R. S. Monteiro, and M.L. Anido

Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro - Brazil

e-mail: mlois@nce.ufrj.br

Abstract

This paper presents a high level language implementation of the Data Encryption Standard (DES) and discusses a design that employs a bit-sliced architecture. The HLL implementation was performed on Borland's® Delphi4™ language and proved to be highly valuable for obtaining the intermediate results that were required for debugging. The key objectives of this work were to make DES available for system applications written in the Delphi4™ language and also to discuss the design of a bit-sliced DES architecture suitable for applications requiring low silicon area.

Keywords: DES, data encryption, ciphering, DES architectures

I. Introduction

For many years, cryptography was the domain of the diplomatic and military world [1,2]. Thanks to the microelectronics revolution, a need for *commercial cryptography* has emerged. This is even more important with the fantastic growth of telephone communications, computer network applications, such as e-commerce, and many other applications that require some sort of security.

Until a few years ago, a 64-bit wide hardware implementation of the DES algorithm [3,4] demanded a considerable amount of hardware, making software implementations more attractive. This is because of the large number of 32-bit load/shift registers, buses and ROMs. Naturally, such hardware demands limited the number of applications that embodied cryptography for security purposes, particularly those applications requiring high-speed and very low hardware cost.

With the advent of microelectronics, some chips have been developed that implement the standard [5,6,7,8]. Many of them have been designed using full-custom or standard cell design approaches and have been implemented using CMOS technology. Silicon Compilation has allowed the description of the DES standard in High Level Languages such as VHDL and to synthesize the description into several technologies, such as standard cell or FPGAs. Microelectronics and Silicon Compilation allow the development of new and more powerful chips supporting highly demanding interactive applications with cryptography support such as real-time video and audio, fast and secure disk access, real-time remote control, etc.

The DES algorithm protects data in two ways. First, *privacy* is protected. After encryption, the sender can be sure that the message, sent over an insecure communication channel such as electronic mail, is only read by the intended receiver. A second and often more important demand is that of *authentication*. After decryption, the receiver can be sure that the message he received came from the original sender and no one else. Both sender and receiver want to be sure that the integrity of the message is guaranteed, i.e., that an opponent did not change, insert, or delete parts of the message.

Section two describes the general characteristics of the DES algorithm by showing block diagrams that represent the basic operations. Section three addresses the implementation of the algorithm in the Delphi™ language. This implementation had two objectives: first, testing the whole algorithm prior to a VHDL description for future synthesis and secondly making DES available for other applications requiring a software implementation (possibly in the form of a DLL or a Delphi's component). Section four discusses a *bit-sliced* implementation of the DES algorithm, which is targeted at systems that have a limited silicon budget and section 5 presents the main conclusions of this work.

II. The DES Algorithm and its Hardware

The DES algorithm was meant to provide cryptographic protection to computer data both in transmission and whilst in storage. As shown in fig. 1, a single DES calculation is a sequence of a 64-bit initial permutation, a consecutive calculation of 16 rounds, and a 64-bit inverse initial permutation. The algorithm passes a block of eight bytes of data, and

takes it through 18 stages of manipulation using substitution and transposition techniques. The data to be encrypted (or decrypted) is controlled by a 56-bit key. Sixteen stages are identical, except that they use 16 different internal *subkeys* derived from the 56-bits of the main key.

Figure 2 illustrates the calculation of the function $f(R, K)$ and contains hardware for one DES round. It consists of 32- and 48-bit modulo2 adders (XOR's Add1 and Add2), eight nonlinear substitution functions with six inputs and four outputs (S boxes), an expansion function E, a permutation function P and two 32-bit registers with support for rotate operations.

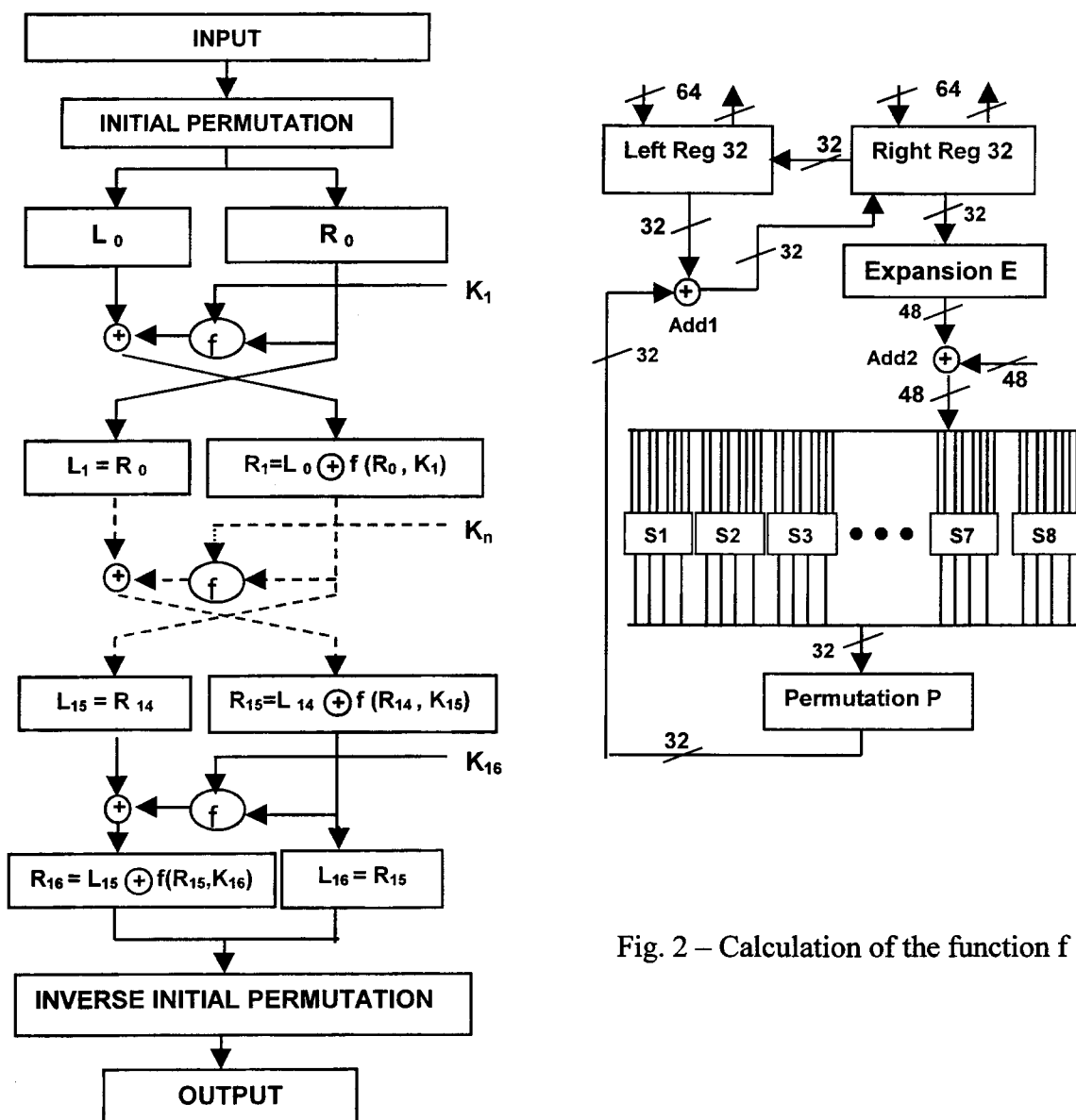


Fig. 2 – Calculation of the function $f(R, K)$

Figure 1 – Major flow of the DES algorithm

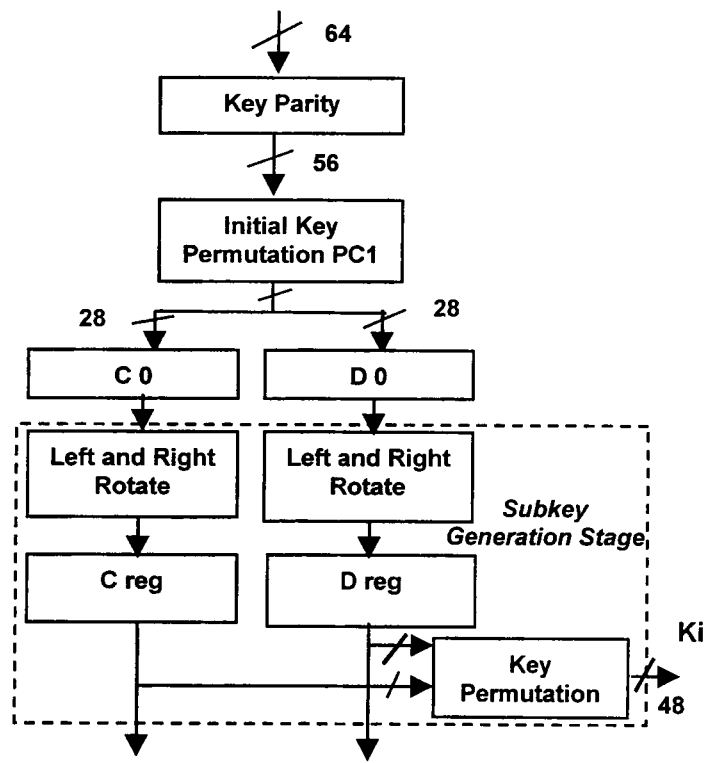


Fig. 3 – Subkey calculation block diagram

Figure 3 depicts the architecture for subkey calculation. For each DES round, a subkey of 48 bits has to be generated. The input key is 64-bit wide and 8 bits are used for parity checking. After an initial key permutation (PC1) the 16 subkeys, one for each round, are derived from the 56-bit key selected for encryption. One subkey is obtained after some left of right rotation and after a 56- to 48-bit permutation and selection.

III. DES Implementation in the Delphi™ Language

When a decision to implement the DES algorithm, in a high level language such as Delphi, was taken, there were two major objectives to be reached. Firstly, it was necessary to test the implementation of the algorithm, obtaining intermediate results that could be used for comparison in a further VHDL bit-slice implementation. Secondly, such implementation can be used in software applications, where the DES algorithm can be adequate. This can possibly be done in the form of a DLL or a Delphi's component.

Figures 4, 5 and 6 illustrate part of the Delphi™ code used to implement the DES algorithm. Figure 7 presents the intermediate results, step by step, for one case example.

```

procedure TfrmDES.btnExecuteClick(Sender: TObject);
const NumberIterations=16;
var
  -----
begin
  for state:=0 to NumberIterations do
    begin
      case state of
        0:
          begin
            load_Registers_Interface(edtInput.text,L,R);
            load_Registers_Interface(edtKey.text,SK1,SK2);
            Invert64Bits(L,R);
            Invert64Bits(SK1,SK2);
            Initial_Permutation(L,R);
          end;
        1..NumberIterations:
          begin
            expand (R,EX_High,EX_Low);
            SUB_SK1 := SK1;
            SUB_SK2 := SK2;
            if (rgpMode.ItemIndex = 0) then
              K(state,SUB_SK1,SUB_SK2)
            else
              K((numberIterations+1)-state,SUB_SK1,SUB_SK2);
            R_aux := R;
            EX_HIGH_S := (EX_HIGH shl 8) or ((EX_LOW and $FF000000) shr 24);
            EX_LOW_S := (EX_LOW and $00FFFFFF);
            Result_P := P(S(SUB_SK1 xor EX_HIGH_S,SUB_SK2 xor EX_LOW_S));
            R := L xor Result_P;
            L := R_aux;
            if state=NumberIterations then
              begin
                permut_final(L,R,Exit_High, Exit_Low);
                Invert64Bits(Exit_High,Exit_Low);
                edtExit.text := inttohex(Exit_High,8) + inttohex(Exit_Low,8);
              end;
            end;
          end;
        end;
      LoadInterface(L,R,EX_High,EX_Low,state);
    end;
  end;

```

Figure 4 – Main procedure of the DES algorithm

```

LeftShifts : array[1..16] of byte = (1,1,2,2,2,2,2,1,2,2,2,2,2,1);
{ K generates a subkey n, where the 64-bit key is represented as :
  - sk1 (32 bits MSB); OUTPUT: -sk1 (24 bits - MSB)
  - sk2 (32 bits LSB);          -sk2 (24 bits - LSB)
}
procedure K(n : byte; var sk1,sk2 : cardinal);
implementation
uses math,uDef;
procedure K(n : byte; var sk1,sk2 : cardinal);
var
  i,j,k,c,d : Cardinal;
begin
  {Performing PC-1 Permutation}
  c := 0;
  d := 0;
  for k:=1 to 2 do           // Reading Vector
    for i:=1 to 28 do
      begin
        if k=1 then
          c := c or (read_bit_word64(sk1,sk2,PC1[k,i])* (1 shl (i-1)))
        else
          d := d or (read_bit_word64(sk1,sk2,PC1[k,i])* (1 shl (i-1)));
        end;
      end;

  for i := 1 to n do
    begin
      for j := 1 to LeftShifts[i] do
        begin
          C := ((read_bit_word32(C,1) shl 27) or (C shr 1)) and $FFFFFFF;
          D := ((read_bit_word32(D,1) shl 27) or (D shr 1)) and $FFFFFFF;
        end;
      end;

    ch1 := 0;
    ch2 := 0;

  {Performing PC-2 Permutation}
  for k:=1 to 2 do // reading vector
    for i:=1 to 24 do
      begin
        if k=1 then
          sk2 := sk2 or (read_bit_word56(d,c,PC2[k,i])*(1 shl (i-1)))
        else
          sk1 := sk1 or (read_bit_word56(d,c,PC2[k,i])*(1 shl (i-1)));
        end;
      end;
    end;
  end.

```

Figure 5 – Procedure for subkey calculation

```

procedure Initial_Permutation (var L,R:LongWord);
var i:LongWord;
    L_aux, R_aux:LongWord;
    Value : LongWord;
begin
    L_aux:=0; R_aux:=0;
    for i:=1 to 64 do
        begin
            value:=read_bit_word4(L,R,array_initial_permutation[i]);
            write_bit_word64(R_aux,R_aux,i,value);
        end;
    end;
    L:=L_aux;
    R:=R_aux;
end;

```

```

procedure expand (R:LongWord; var EX_High, EX_Low: LongWord);
var i,value:LongWord;
begin
    EX_High := 0; EX_Low := 0;
    for i:=1 to 48 do
        begin
            valor := read_bit_word32(R,array_expansao[i]);
            write_bit_word64(EX_High,EX_Low,i,value);
        end;
    end;

```

```

procedure final_Permutation(L,R:LongWord;var Out_High, Out_Low :
LongWord);
var i: LongWord;
    value : LongWord;
begin
    Out_High:=0; Out_Low:=0;
    for i:=1 to 64 do
        begin
            value := read_bit_word64(L,R, array_final_permutation [i]);
            write_bit_word64(Out_High,Out_Low,i,valor);
        end;
    end;

```

Fig. 6 – Some additional procedures: expansion, initial and final permutations

Data Encryption Standard - DES

☐ INPUT
 ☐ KEY
 ☐ OUTPUT

80856B0B9CEEE642

7CA110454A1A6E57

9D924624B3F2B4A0

☒ Encryptar
 ☐ Decryptar

☒ RUN

L0	704E0827	R0	373C26CE	E1	00001AE9F810D65C
L1	373C26CE	R1	698E2153	E2	0000B53C5C102AA6
L2	698E2153	R2	7A67938F	E3	0000BF430FCA7C5E
L3	7A67938F	R3	0727680F	E4	000080E90EB5005E
L4	0727680F	R4	B6B5D099	E5	0000DAD5ABEA14F3
L5	B6B5D099	R5	3F3C40DF	E6	00009FE9F82016FE
L6	3F3C40DF	R6	F3E2E2EE	E7	00007A7F0570575D
L7	F3E2E2EE	R7	89D9F55A	E8	0000453EF3FAAAF5
L8	89D9F55A	R8	CC16668C	E9	00006580AC30D459
L9	CC16668C	R9	043EA8F6	E10	00000081FD5517AC
L10	043EA8F6	R10	F66AC8BA	E11	00007AC3556515F5
L11	F66AC8BA	R11	367FC640	E12	00001AC3FFE0C200
L12	367FC640	R12	22034F46	E13	0000104006A5EA0C
L13	22034F46	R13	BC5DEA31	E14	0000DF82FBF541A3
L14	BC5DEA31	R14	690C79E6	E15	00003528583F3F0C
L15	690C79E6	R15	6C801FCF	E16	0000B594000FFE5E
L16	6C801FCF	R16	88B2CE24		

Fig. 7 – Intermediate results (Li, Ri and Ei) for one combination of input data and key.

IV. A bit-slice architecture for the implementation of the DES algorithm

Despite the enormous increase in silicon density in recent years, a 64-bit wide implementation of the DES algorithm still takes a considerable silicon area. However, there are many applications where data encryption is just one small part of a much larger system. In these cases, silicon area is a precious resource and has to be used judiciously. This section discusses a bit-slice (or nibble) implementation of the DES algorithm.

In the bit-sliced architecture illustrated by figure 8, each byte of the data is loaded into the L & R registers – a nibble in L and a nibble in R. As soon as the 64-bit (8 registers of 4 bits each) data is loaded into L&R, and the 64-bit key into the subkey generator, encryption (decryption) begins by operation on a nibble (4 bits) at a time in a nibble serial manner[9,10], that is, the registers operate as a cyclic register chain. The key is loaded only

once, and then remains in the subkey generator throughout the process. The encrypted data appears at the outputs of the L & R registers after every 186 cycles. It is worth noting that the loading of the data input and the unloading of the finished output can be pipelined if a separate data bus is provided (178 cycles are required in this case). Using the same data bus to load data and key, it is necessary to load the key first. Additionally, as the data is manipulated in two blocks of 32 bits, each operation involving 32 bits requires eight cycles (8 x 4-bits). The overall operation of the architecture is:

1. Load eight bytes of main key into the subkey generator.
2. Load eight bytes of data into the L & R registers.
3. Allow 2 wait-states to line up nibbles inside L&R.
4. Repeat 16 times:
 - 2 cycles of invalid E-bits,
 - 8 cycles to generate 8 sets of E-bits (each 6-bits wide).

This will allow the permuter to generate the correct P-bits and hence perform 15 stages of F (R_n , K_n).

Thus, $L_0R_0 \rightarrow L_{15}R_{15}$.

5. Eight additional cycles to generate the next block of E-bits, and takes $L_{15}R_{15}$ to $L_{16}R_{16}$.
6. Eight cycles to unload the encrypted/decrypted data.

During the first subkey cycle, step (4) above, the first ten cycles do not operate on the L&R registers as the permuter has not yet been loaded. However, in subsequent cycles, each iteration of step (4) takes L_nR_n to $L_{n+1}R_{n+1}$.

At step (6), the next eight bytes of data can be loaded simultaneously if there are two data-buses, in which case proceed to step (3). If not, continue from step (2).

To load the key, the data, and operating on them to produce eight bytes of encrypted/decrypted output will require 194 clock cycles. However, the time required to just encrypt (or decrypt) the data is 168 cycles. This is equivalent to 1.68 μ s with a 100 MHz clock.

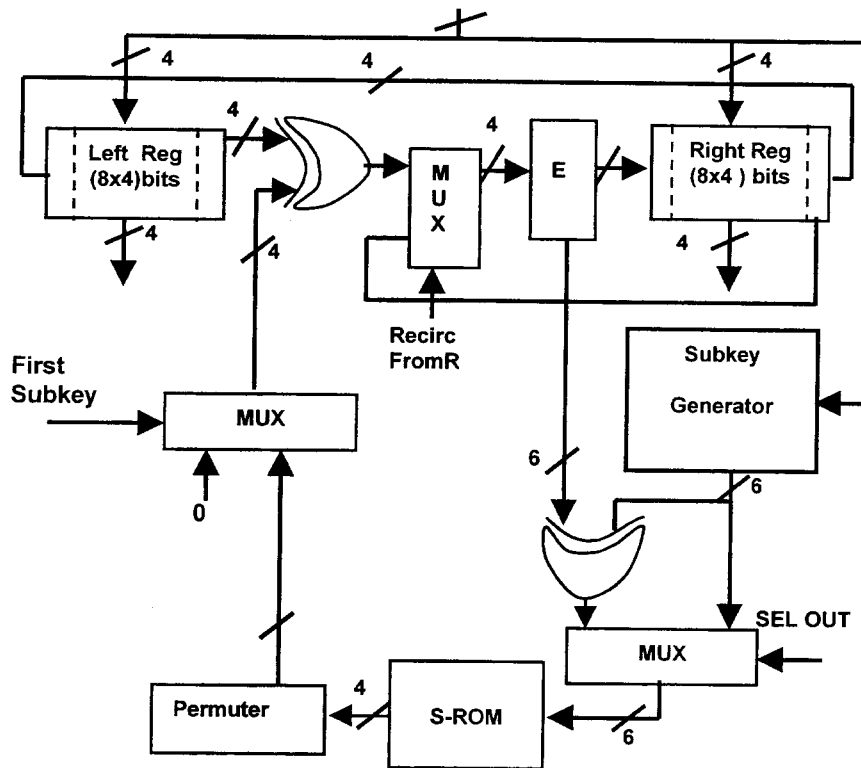


Fig. 8 – Bit-sliced architecture for DES implementation

V. Conclusions

This paper described a high level language implementation of the Data Encryption Standard (DES) in the Delphi™ language. This implementation had two objectives: first, testing the whole algorithm prior to a VHDL description for future synthesis and secondly making DES available for other applications requiring a software implementation (possibly in the form of a DLL or a Delphi's component).

This paper also focused on the design description of a bit-slice architecture (using 4-bit nibbles) that can be used when there are silicon area constraints. This is the case in many situations where the ciphering/deciphering of data is just one small part of a much larger problem and silicon area has to be used judiciously. The designer has to balance silicon area and performance and a 8-bit nibble can provide a better performance, but at a higher hardware cost.

E. Hellman, "Privacy and authentication: An introduction to
EE, col. 67, no. 3, pp. 397-427, Mar. 1979.

uction to Cryptology and PC Security", McGraw Hill Book Co.,

andard, Federal Information Processing Standard (FIPS) 46, Nat.

eration, Federal Information Processing Standard (FIPS) 81, Nat.

medt, J. Goubert, F. Hoonaert and J. J. Auisquater, "Efficient
implementations of the DES" in advances in Cryptology, Proc.

. Hoornaert, J. Vandewalle and H. J. de Man, "Security and
on of a New DES Data Encryption Chip", IEEE Journal of Solid-
No. 3, June, 1988.

gle chip encrypts data at 14 Mb/s", Electronics, vol. 54, pp. 161-

Matusevich, and J. Plany, "An LSI digital encryption processor
Mag. Vol. 23, no. 7, pp. 30-41, July 1985.

. Zaluska, "A VLSI DES Implementation: Subkey Generation",
Electronics and Computer Science, Univ. of Southampton, UK,

J. Zaluska, "Part Implementation of a Data Encryption Standard
thesis, Dept of Electronics and Computer Science, Univ. of

[1] W. Diffie and M.
cryptography," Proc. IE

[2] B. Beckett, "Introdu
2000

[3] Data Encryption Sta
Bur. Stand., Jan. 1977.

[4] DES Modes of Op
Bur. Stand. Dec. 1980.

[5] M. Davio, Y. Des
hardware and software
Crypto 84, Aug. 1984.

[6] I. Verbauwhede, F
Performance Optimizati
State Circuits, Vol. 23, N

[7] D. MacMillan, "Sin
165, June, 16, 1981.

[8] R. C. Fairfield, A.
(DEP), "IEEE Commun.

[9] C. Mistry and E. J
M.Sc. thesis, Dept. of
1987.

[10] H. S. Gill and E. J
Chip Set", M. Sc. Th
Southampton, UK, 1987